HARDWARE, SOFTWARE AND PEOPLEWARE

UNISA

SAICSIT 2001

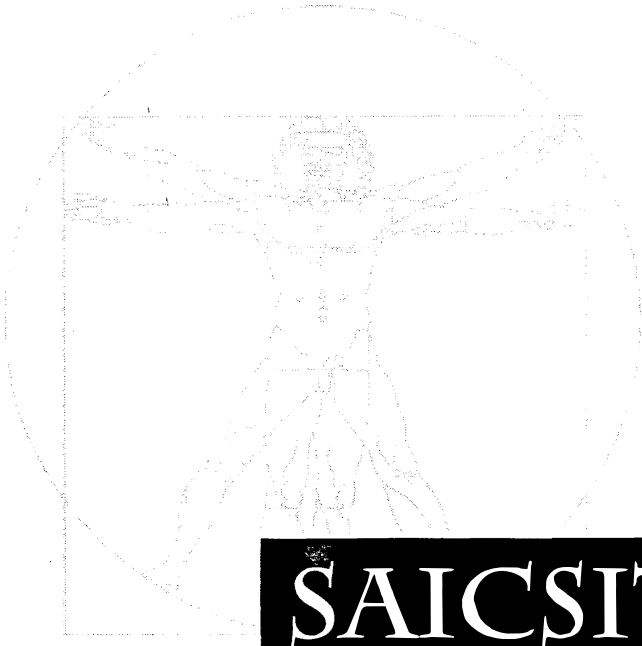
Edited by
Karen Renaud
Paula Kotzé
Andries Barnard

HARDWARE, SOFTWARE AND PEOPLEWARE

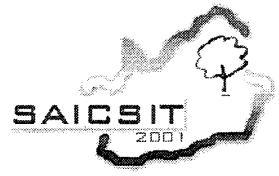
South African Institute of Computer Scientists and Information Technologists Annual Conference

25 – 28 September 2001

Pretoria, South Africa



SAICSIT 2001



Edited by Karen Renaud, Paula Kotzé & Andries Barnard

University of South Africa, Pretoria

Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists

First Edition, First Impression
ISBN: 1-86888-195-4

Published by Unisa Press
University of South Africa
P O Box 392, Pretoria, 0003

Cover Design by Tersia Parsons

Editors: Karen Renaud, Paula Kotzé & Andries Barnard

Electronic Publication by the Editors

Printed by Unisa Press
2001

Table of Contents

Abstracts of Electronic Papers

Message from the SAICSIT President

The South African Institute of Computer Scientists and Information Technologists (SAICSIT) was formed in 1982 and focuses on research and development in all fields of computing and information technology in South Africa. Now in the 20th year of its existence, SAICSIT has come of age, and through its flagship series of annual conferences provides a showcase of not only the best research from the Southern-African region, but also of international research, attracting contributions from far afield. SAICSIT does, however, not exist or operate in isolation.

More than 50 years have passed since the first electronic computer appeared in our society. In the intervening years technological development has been exponential. Over the last 20 years there has been a vast growth and pervasiveness of computing and information technology throughout the world. This has led into the expansion and consolidation of research into a diversity of new technologies and applications in diverse cultural environments. During this period huge strides have also been made in the development of computing devices. The processing speed of computers has increased thousand-fold and memory capacity from megabytes to gigabytes in the last decade alone. The Southern African region did not miss out on these developments.

It is hardly possible for such quantitative expansion not to bring a change in quality. Initially computers had been developed mainly for purposes such as automation for the improvement of processing, labour-reduction in production and automation control of machinery, with artificial intelligence, which made great strides in the 1980s, seen as the ultimate field to which computers could be applied. As we moved into the 1990s it was recognized that such an automation route was not the only direction in the improvement of computers. The expansion of processing power has enabled image data to be incorporated into computer systems, mainly for the purpose of improving human utilisation. For most computer technologies of the 1990s, including the Internet and virtual reality, automation was not the ultimate purpose. Humans were increasingly actively involved in the information-processing loop. This involvement has gradually increased as we move into the 21st century. Development of computer technology based not on automation, but on interaction, is now fully established.

The method of interaction has significantly changed as well. The expansion of computer ability means that the same function can be performed far more cheaply and on smaller computers than ever before. The advent of portable and mobile computers and pervasive computing devices is ample evidence of this. The need for users to be at the same location as a computer in order to reap the benefits of software installed on that computer is becoming an obsolete notion. Time and space are no longer constraints. One of the most discussed impacts of computing and information technology is *communication* and the easy accessibility of information. This changes the emphasis for research and development – issues such as cultural, political, and economic differences must, for example, be accommodated in ways that researchers have not previously considered. Our goal should be to enable users to benefit from technological advances, hence matching the skills, needs, and expectations of users of available technologies to their immense possibilities.

The conference theme for the SAICSIT 2001 Conference – *Hardware, Software and Peopleware: The Reality in the Real Millennium* – aims to reflect technological developments in all aspects related to computerised systems or computing devices, and especially reflect the fact that each influences the others.

Not only has SAICSIT come of age in the 21st century, but so has the research and development community in Southern Africa. The outstanding quality of papers submitted to SAICSIT 2001, of which only a small selection is published in this collection, illustrates both the exciting and developing nature of the field in our region. I hope that you will enjoy SAICSIT 2001 and that it will provide opportunities to cultivate and grow the seeds of discussion on innovative and new developments in computing and information technology.

Paula Kotzé
SAICSIT President

Message from the Chairs

Running this conference has been rewarding, exciting and exhausting. The response to the call for papers we sent out in March was overwhelming. We received 64 paper submissions for our main conference and twelve for the postgraduate symposium. We had a panel of internationally recognized reviewers, both local and international. The response from the reviewers was impressive – accepting a variety of papers and *mostly* returning the reviews long before the due date. We were struck, once again, by the sheer magnanimity of academia – as busy as we all are, we still manage to contribute fully to a conference such as SAICSIT.

After an exhaustive review process, where each paper was reviewed by at least three reviewers, the program committee accepted 26 full research papers and 14 electronic papers. Five papers were referred to the postgraduate symposium, since they represented work in progress – not yet ready for presentation to a full conference but which nevertheless represented sound and relevant research. The papers published in this volume therefore represent research of an internationally high standard and we are proud to publish it. Full electronic papers will be available on the conference web site (http://www.cs.unisa.ac.za/saicsit2001/).

Computer Science and Information Systems academics in South Africa labour under difficult circumstances. *The popularity of IT courses stems from the fact that IT qualifications are in high demand in industry, which leads in turn to a shortage of IT academic staff to teach the courses, even when posts are available. The net result is that fewer people teach more courses to more students. IT departments thus rake in ever-increasing amounts of state subsidy for their universities. These profits, euphemistically labelled "contribution to overhead costs", are deployed in various ways: cross-subsidization of non-profitable departments; maintenance of general facilities; salaries for administrative personnel, etc. Sweeteners of generous physical resources for the IT departments may be provided. We have yet to hear of a University in South Africa where significant concessions have been made in terms of industry-related remuneration. At best, small subventions are provided. As a result, shortages of quality staff remain acute in most IT departments – especially at senior teaching levels. What is even worse is that academics in these departments have to motivate the value of their conference contributions and other IT outputs to selection committees, often dominated by sceptical academic power-brokers from the more traditional departments whose continued survival is underwritten by IT's contribution to overhead costs.*[1]

The papers published in this volume are conclusive evidence of the indefatigability and pertinacity of Computer Science and Information Systems academics and technologists in South Africa. We are proud to be part of such a prestigious and innovative group of people.

In conclusion, we would like to thank the conference chair, Prof Paula Kotzé, for her support. We also specially thank Prof Derrick Kourie for his substantial contribution. Finally, to all of you, contributors, presenters, reviewers and organisers – a big thank you – without you this conference could not be successful.

Enjoy the Conference!
Karen Renaud & Andries Barnard

[1] This taken almost verbatim from Professor Derrick Kourie's SACLA 2001 paper titled: "*The Benefits of Bad Teaching*".

Conference Organisation

General Chair
Paula Kotzé

Programme Chairs
Karen Renaud
Andries Barnard

Organising Committee Chairs
Lucas Venter, Alta van der Merwe

Art and Design
Tersia Parsons

Sponsor Liaison
Paula Kotzé, Chris Bornman

Secretarial & Finances
Christa Prinsloo, Elmarie Havenga

Marketing & Public Relations
Klarissa Engelbrecht, Elmarie van Solms, Adriaan Pottas, Mac van der Merwe

Audio Visual
Tobie van Dyk, Andre van der Poll, Mac van der Merwe

Program Committee

Bob Baber – McMaster Univeristy, Canada
Andries Barnard – University of South Africa
Judy Bishop – University of Pretoria
Andy Bytheway – University of the Western Cape
Andre Calitz – University of Port Elizabeth
Elsabe Cloete – University of South Africa
Carina de Villiers – University of Pretoria
Alan Dix – Lancaster University, United Kingdom
Jan Eloff – Rand Afrikaans University
Andries Engelbrecht – University of Pretoria
Chris Johnson – University of Glasgow, United Kingdom
Paul Licker – University of Cape Town
Paula Kotzé – University of South Africa
Derrick Kourie – University of Pretoria
Philip Machanick – University of the Witwatersrand
Gary Marsden – University of Cape Town
Don Petkov – University of Natal in Pietermaritsburg
Karen Renaud – University of South Africa
Ian Sanders – University of the Witwatersrand
Derrick Smith – University of Cape Town
Harold Thimbleby – Middlesex University, United Kingdom
Theda Thomas – Port Elizabeth Technikon
Herna Viktor – University of Pretoria, South Africa
Bruce Watson – Universities of Pretoria and Eindhoven
Janet Wesson – University of Port Elizabeth

Referees

Molla Alemayehu
Trish Alexander
Adi Attar
Bob Baber
Andries Barnard
John Barrow
Judy Bishop
Gordon Blair
Arina Britz
Andy Bytheway
André Calitz
Charmain Cilliers
Elsabe Cloete
Gordon Cooper
Richard Cooper
Annemieke Craig
Thad Crews
Quintin Cutts
Michael Dales
Carina de Villiers
Alan Dix
Dunlop Mark
Elize Ehlers
Jan Eloff
Andries Engelbrecht
Klarissa Engelbrecht
David Forsyth
John Galletly
Vashti Galpin
Wayne Goddard
Alexandré Hardy
Scott Hazelhurst
Johannes Heidema
Tersia Hörne
Chris Johnson
Bob Jolliffe
Paula Kotzé
Derrick Kourie
Les Labuschagne
Paul Licker
Philip Machanick
Anthony Maeder
David Manlove
Gary Marsden
Thomas Meyer
Elsa Naudé
Martin Olivier
Don Petkov
Pekka Pihlajasaari
Nelisha Pillay
Laurette Pretorius
Karen Renaud
Ingrid Rewitzky
Sheila Rock
Markus Roggenbach
Ian Sanders
Justin Schoeman
Martie Schoeman
Elsje Scott
Derek Smith
Elmé Smith
Adrie Stander
Harold Thimbleby
Theda Thomas
Judy Van Biljon
Alta Van der Merwe
André van der Poll
Tobias Van Dyk
Lynette van Zijl
Lucas Venter
Herna Viktor
Bruce Watson
Janet Wesson

Conference Sponsors

Keynote Abstracts

JavaCloak — Considering the Limitations of Proxies for Facilitating Java Runtime Specialisation

Karen Renaud

University of South Africa, Muckleneuk Ridge, Pretoria, South Africa,
renaukv@unisa.ac.za

Abstract

This paper discusses issues pertaining to mechanisms which can be used to change the behaviour of Java classes at runtime. The proxy mechanism will be compared to and contrasted with other standard approaches to this problem. Some of the problems the proxy mechanism is subject to will be expanded upon. The question of whether statically-developed proxies are a viable alternative to bytecode rewriting was investigated by means of the JavaCloak system, which uses statically-generated proxies to alter the run-time behaviour of externally-developed code. The issues addressed in this paper include ensuring type safety, dealing with the self problem, object encapsulation, and issues of object identity and equality. Some performance figures are provided which demonstrate the load the JavaCloak proxy mechanism places on the system.
Keywords: *Reflection, Java, Proxies, JavaCloak, Runtime Specialisation.*
Computing Review Categories: *D.1.5, D.3.3*

1 Introduction

There is often a need to specialise the runtime behaviour of classes long after implementation of the classes. There are a number of non-functional requirements which could require such specialisation such as, for example, security, system instrumentation and distribution. Post-implementation specialisation can be achieved by:

1. Tailoring source code [20];
2. Providing a customised Java Virtual Machine (JVM) [8, 10, 15];
3. Using wrapper/proxy objects. The wrapper/proxy pattern was identified by Gamma *et al* in their seminal book on design patterns [9]. Dynamically-generated proxies were used by Welch and Stroud in their *Dalang* system [17];
4. Bytecode Engineering:
   (a) Pre-Load-time — Providing bytecode-rewriting tools to allow programmers to make changes to class bytecode [4, 6].
   (b) Load-time — Providing a custom `classloader`[1] which integrates a meta-object protocol with the original bytecode. These approaches are illustrated in the *Javassist* and *Kava*[2] systems [2, 19].

[1] A Java system class which loads classes as required by an application. The JVM allows a programmer to extend this class to specialise its behaviour.

[2] *Kava* is the successor of *Dalang*, and was developed once the limitations of the *Dalang* approach became apparent.

Each of the above approaches has both advantages and disadvantages. Section 2 will discuss the pros and cons of these mechanisms. Many of the approaches have been tried and tested but the effects of one, namely that of statically-generated proxies, has not yet been quantified. The research reported in this paper was done to investigate both the viability of using statically-generated proxies for runtime specialisation, and the suitability of Java for expediting such specialisation. To this end the *JavaCloak* system was developed, using Java. JavaCloak uses statically-developed proxies to alter system runtime behaviour. The system is described in Section 3. Section 4 discusses the problems JavaCloak encountered — related to the use of proxies. Section 5 presents the results of performance measurements with JavaCloak proxies. Section 6 concludes.

2 Runtime Specialisation

Source-code tailoring cannot really be considered to be a runtime specialisation technique although it *is* often used as a mechanism for post-implementation adjustment of behaviour. Source-code tailoring will always be an inelegant and untenable solution to the problem because programmers may introduce new errors into the program, change the behaviour of the program, or omit to insert code consistently throughout the program.

Customised JVMs provide a better solution than source-code tailoring but are often not a viable option due to their being non-standard, and often tightly

linked to one specific platform.

The latter runtime specialisation mechanisms, *bytecode engineering* and *proxies*, will be discussed in the following two sections.

2.1 Bytecode Engineering

Some researchers have investigated tools and techniques that allow Java bytecode to be changed without needing access to the source code. These tools can be divided into two categories, those that support *bytecode rewriting* [4, 6] and those that provide a *meta-object protocol* [18] and dynamically adapt code at runtime [11].

The bytecode-rewriting tools in the first category are very useful as they allow programmers to change a class definition to meet a local need. For example, the bytecode of a class can be changed to ensure that it is compatible with the Java Object-Serialization mechanism so that instances of the modified class can be passed across a network or written to disk.

However, bytecode rewriting has a number of disadvantages. The changes must be applied every time a class requiring it is recompiled (to ensure the rewriting has been correctly applied to the newly generated bytecode) and there is the extra effort of determining whether a new version requires amending or not.

In addition, systems that require bytecode to be rewritten, such as ObjectStore's PSE Pro [5], may burden the programmer with the management of two sets of classes. For example, one class (`A`) may depend on another class (class `B`) that has to be post-processed[3]. Class `B` should therefore be compiled first and post-processed before class `A` is compiled. Tracking these kinds of relationships for significant bodies of code is a non-trivial problem. This kind of bytecode rewriting is simply performed at too low a level of abstraction. A higher level of abstraction is required and thus the meta-object protocol approach has been developed [19].

Meta-object protocols [12] are a powerful programming paradigm for associating new behaviour with a program. The authors of [18] have defined a meta-object protocol for Java that rewrites bytecode at load time. The Kava approach allows either the addition of new behaviour to the class or the ability to selectively override invocations on a method-by-method basis [18]. Kava also allows exception handling to be intercepted so that exceptions can be overridden.

Javassist [2] is another a bytecode rewriting tool based on structural reflection[4]. It also makes use of a meta-object protocol. However, it does not support reflection on methods inherited from superclasses [19] — a problem that has been resolved by Kava.

[3] i.e., its bytecode changed by another tool after class `B` has been compiled

[4] Structural reflection allows a program to change the definition of a class, a function or a record, for example.

The system described by Keller and Hölzle in [11] is targeted at integrating Java classes with other, non-compatible, classes. This system is based on the dynamic adaptation of bytecode, as the class is loaded into the Java virtual machine. The programmer identifies which class should have its bytecode modified to ensure that the classes can interoperate. Their system ensures that incompatible classes can be used together by adding, renaming or removing methods, or by changing the class hierarchy.

Dynamic bytecode rewriting imposes a runtime overhead when loading classes. This overhead could be reduced by performing the post-processing once, statically, before the program is run. However, all the classes that the program could possibly use would have to be identified and it would have to be re-applied every time the related meta-object is changed.

2.2 Proxies

The definition of proxy[5] used in this paper accords with that of Gamma *et al* [9]. Proxies may augment, or report on, the behaviour of the original classes. Most runtime specialisation is done in order to add non-functional properties to the system, so that the functionality of the original class will often still be required. Thus proxies will probably be required to invoke methods on instances of the original, matching classes. Proxies can be set up at different times:

- *Compile-time* — such proxies are generated statically, and compiled and then loaded by the JVM instead of the original classes.
- *Load-time* — such proxies are generated on-the-fly by providing the JVM with a customised `classloader`. This `classloader` generates the wrapper and provides it in place of the original class [17]. The substitution is thus done when the application requests an instance of a class that is being wrapped, causing a different definition of the class to be loaded.
- *Runtime* — this can only be done by means of a reflective JVM [14], which allows the already-loaded bytecode to be altered so that a proxy can be substituted for the original class.

The Java 2 platform (version 1.3) defines dynamic `Proxy` classes [1]. These classes implement a list of interfaces invoked at runtime when the class is created. However, a `Proxy` class has a number of undesirable properties from the point of view of runtime specialisation[6]:

1. Dynamic proxy classes are `public`, `final` and cannot be `abstract` — which means that they cannot form part of an inheritance structure;

[5] The terms "proxy" and "wrapper" are synonyms in this paper.

[6] It may be possible to work around a number of these issues.

2. The proxy class must extend `java.lang.reflect.Proxy` — once again causing problems with inheritance;

3. A programmer-defined invocation handler must be used to dispatch the method invocation to the wrapped instance at runtime — requiring the skills of an expert programmer; and

4. The Java security domain of the proxy is the same as that of system classes loaded by the bootstrap `classloader`, such as `java.lang.Object`, because the code for the proxy class is generated by trusted system code. This security domain is typically granted the Java security permission `java.security.AllPermission`, and so these proxy classes are effectively not restricted by the Java 2 security mechanism.

Given the above limitations of Java-provided proxies, an alternative approach was explored, which exploits statically-generated proxy classes. The programmer can customise the generated proxy source code so as to change the runtime behaviour of the class that the proxy class wraps up the original class. This system is called JavaCloak [16] and was developed in order to investigate both the viability of such a scheme as well as to determine the performance penalty associated with statically-generated proxies.

JavaCloak has worked well for simple cases. However, Java's runtime typing rules and other facilities, such as the capabilities of the `java.lang.reflect` package, prevent us from providing as general a model as possible.

3 The JavaCloak System

The problems with load-time generated proxies are well known [17]. However, statically-generated proxies have not been tried and tested, due to the fact that the JVM will not load proxies in place of the original classes unless they have the same names as the original classes, and if they have the same names proxy objects cannot access instances of the original classes and so proxies are unable to delegate method invocations. In order to test statically-generated proxies two problems have to be overcome:

1. How does one get the JVM to load the proxy classes instead of the original classes when an instance of the wrapped class is requested by the application — if instances of both classes need to be created at runtime (in order to allow delegation of method invocations to instances of the original classes)?

2. If one overcomes the previous problem, how does one then access the original classes from within the proxies — in the light of the fact that the proxies and the original classes have the same name? The JVM generally expects a class definition to be unchanging. It will routinely generate a `ClassCast` exception if it encounters an instance of a class that differs, in definition, from other instances of the same class.

The JavaCloak system has solved the former problem by altering the `CLASSPATH` at runtime and the latter by using a customised `classloader` at runtime. The proxy objects are statically and automatically generated, by using the Java reflection package (`java.lang.reflect`) to mediate access to the original objects — and are produced in the form of Java code.

The required behavioural changes can be incorporated into this code by the programmer — a process that requires no additional programmer skills[7]. The programmer is free either to augment the classes with reporting facilities in the very simplest case, or to change the behaviour of the methods completely by invoking a method on an instance of another class altogether, or on a remote object.

After the customisation of the proxy source code has been performed, and they have been compiled, instances of the *proxy* objects are created at runtime when the application instantiates an instance of the original class. The proxy code then delegates all calls made on its instance's `public` methods to equivalent methods defined on the instance of the original class. The process becomes more complex when parameters and return values are involved — hence the inclusion of the JavaCloak runtime manager.

The key enabling features of JavaCloak, shown in Figure 1, will be described below:

1. *Proxies* — The pre-runtime generation of proxies should ensure that the runtime penalty of using JavaCloak is minimised and secondly that the Java programmer can specialise the runtime behaviour of the class in a finely-grained and flexible manner.

2. *Manipulation of the `CLASSPATH` at runtime* — the location of the proxy classes is inserted into the `CLASSPATH` *instead of* the location of the original classes. The location of the original classes is then provided for use by the JavaCloak `classloader` by means of a runtime system property setting.

3. *A customised `classloader`* — JavaCloak makes use of a specially defined `classloader`, embedded within the runtime manager, to load the original classes at runtime. This `classloader` loads the original definition of the class from a

[7] This could also be seen as source-code tailoring but it should be borne in mind that this is much simpler to do since the forwarding model is very simple and the programmer cannot introduce errors into the original code or change the behaviour of that class in any way.

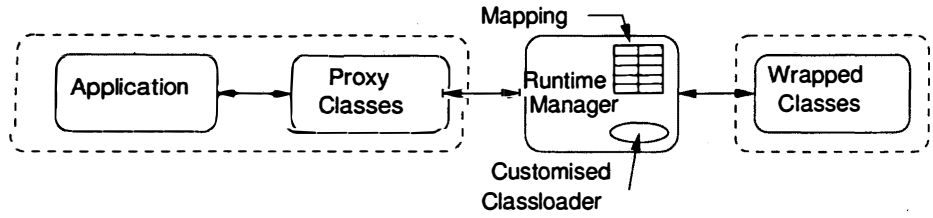


Figure 1: JavaCloak Architecture

location supplied to the `classloader` by a runtime variable when the application is executed. The JVM tests type-equivalence by comparing both the class name and the instance of the `classloader` that loaded the class. Thus if two different `classloaders` are used to load classes with the same name the JVM does not consider the classes to be identical.

4. *The JavaCloak runtime manager* — this manager is the key to JavaCloak's extra level of abstraction. The runtime manager encapsulates the `classloader` and loads the original classes when requested by the proxy objects. It also maintains a mapping between proxies and their matching original objects so that any parameters passed between the two can be translated as required.

   For instance, say a method invocation passes an instance of a proxy class as a parameter. The proxy class has no meaning to the original class and, if passed to it, would cause the system to generate an exception. The runtime manager offers a facility for substituting original objects so that such parameters are 'unwrapped' before they are passed to the method in the original class.

   In the same way the original object may pass a reference to a wrapped object back to the proxies. The runtime manager offers a facility to substitute the proxy for such objects again so that they do not cause `ClassCast` exceptions in the application (which has no concept of the original classes).

The structure of interaction between the application, proxy and JavaCloak runtime manager is shown in Figure 2. When the proxy is instantiated it asks the runtime manager to instantiate an instance of the original object. The runtime manager then:

1. uses the customised `classloader` to load the original class definition from the location specified in the runtime variable.
2. creates an instance of the original class.
3. inserts an entry into the mapping table linking the proxy to the original object.
4. returns a reference to the original object to the proxy.

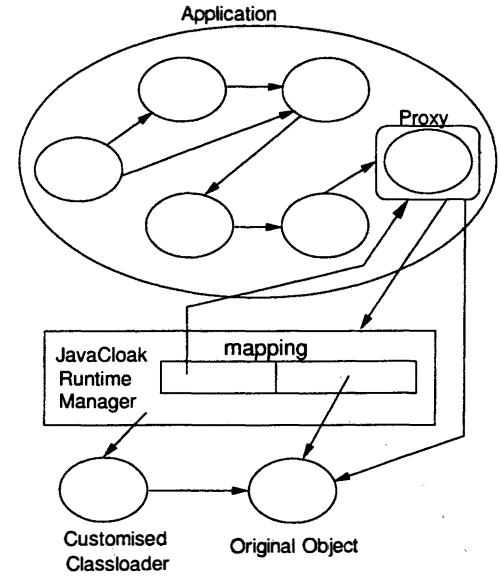


Figure 2: Interaction at Runtime

When a method is invoked on the proxy the following occurs:

1. If the method has parameters and the parameter objects are references to proxies then the proxy asks the runtime manager for references to the original objects.
2. The proxy invokes the method on the original object.
3. If the method returns a value the proxy receives the return value and stores it in a variable of type `Object`:
   (a) If the return value is an instance of a wrapped class then the proxy asks the runtime manager for a reference to the matching proxy object and the proxy returns the reference to the proxy object to the application.
   (b) Otherwise the return value is 'casted' to the correct type and returned to the application.

JavaCloak has been used successfully to wrap classes and to report on access to instances of these classes. The following section will discuss some problems en-

countered with the use of JavaCloak proxies for reflection[8].

4 Proxy-Related Problems

The proxy approach is not without its problems. One JavaCloak-specific problem is that JavaCloak makes use of manipulation of the `CLASSPATH` in order to divert the JVM to the proxy classes rather than the original classes. This obviously won't work for system classes which are loaded by the JVM automatically and not by means of a search of the `CLASSPATH`. This problem could be alleviated by the provision of a customised `classloader` to the JVM. Other generic proxy-enabled reflection-related problems will be discussed in the following subsections.

4.1 Type Safety

It is important for the smooth functioning of a system incorporating JavaCloak proxies that the proxies be type-equivalent to the original classes. Therefore the generated proxy must have the same fully-qualified class name as the original class and it must be loaded at runtime by the correct instance of the `classloader`. This ensures that the application can use the proxy as if it were the original class, and the inheritance structure is not broken. There are some tricky problems related to this apparent transparent substitution, some of which have been solved, and others of which remain.

4.1.1 Maintaining Two Identical Class Names

One runtime problem that JavaCloak had to overcome is that the JVM needs to have instances of both classes within the system at the same time. One cannot load two different definitions of the same class name into the JVM without some special mechanism. The only way to achieve this apparent conflict is by means of the use of a different `classloader` for each class. This is because the JVM tests type equivalence by testing both the class name, and the `classloader` that loaded the class definition. If the two classes are loaded by two different `classloaders` the JVM considers them to be different classes — even though the fully-qualified class names are the same.

Once the two classes are in the JVM it is essential that instances of the two classes be kept strictly separate. If instances of these two classes are used in the same piece of code, e.g., one instance is passed to one of the methods of the other instance, a class-cast exception will be raised. The JavaCloak runtime manager is used to maintain a strict separation between the two types of objects. This is achieved by only mentioning the type of the original class as a Java `String` and manipulating it via the reflection mechanism defined in `java.lang.reflect`.

4.1.2 Public fields

It is possible for the original definition of a class to contain non-static `public` fields. If this is the case, to ensure correctness it must be possible for the field to be accessed directly from the application — and not only via programmer-provided `set` and `get` methods. Unfortunately Java does not model field access as method invocation, so there is no opportunity to redirect accesses to the `public` fields via the proxy if the application accesses the fields of the original class directly. Given the current definition of Java, providing the application with access to `public` fields consistently and transparently in the presence of JavaCloak proxies is not possible[9].

In following the proxy approach, we need to either implement a system of 'watchers' for each `public` field, or assume a clean object-oriented programming model where all field accesses are controlled by means of suitable method calls that can then be used to forward the call to the original object. There is no support in Java to be able to implement the 'watchers', thus the only approach is to assume (and limit the programmer) to a clean object-oriented programming model.

4.1.3 Inheritance

This is a tricky problem for JavaCloak. If JavaCloak provides a proxy for class B, which inherits from class A, which also has a proxy, it is important for there not to be multiple links between the proxy *world* and the wrapped *world*. The need to keep these two *worlds* apart was alluded to in Section 4.1.1. JavaCloak must ensure that proxy A's constructor does not request the runtime manager to create a matching original object if it is invoked from B's constructor.

To illustrate this point, consider the situation as shown in Figure 3. We have a class `V`, which is extended by a class `FSV`, in turn extended by class `DFSV`. The application instantiates an instance of `DFSV`. Since the system loads the JavaCloak proxy rather than the original class, `DFSV` is loaded. Since `FSV` and `V` are also wrapped, `DFSV`'s constructor will also create an instance of both `FSV` and `V`. If each constructor automatically creates an instance of the matching original class, the links will be as shown on the lefthand side of the diagram.

[8]The term 'reflection' is generally used to denote the ability of a system to change its behaviour by examining certain properties at runtime and adapting the system depending on these properties. Some authors would refer to JavaCloak's activities as reflection, and others would refer to it as runtime specialisation. The authors of [17] use the term *reflection* to refer to their proxy-based approach, and their example will be followed from here onwards.

[9]This argument also applies to fields marked `protected` and those that are scoped at the package level in Java.

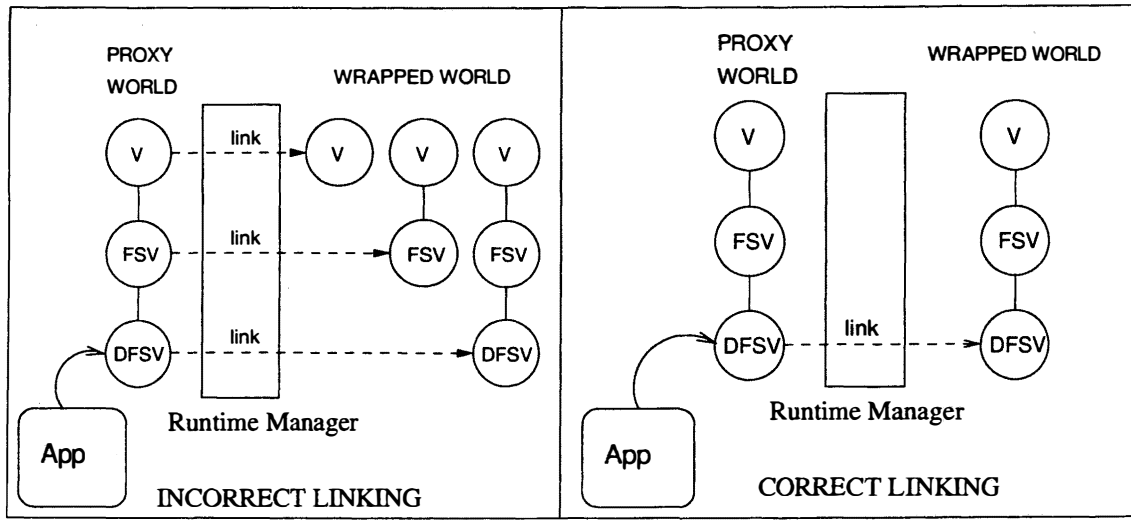


Figure 3: Inheritance across Worlds

The multiple links will cause havoc when proxy objects invoke superclass methods, because any changes made by methods invoked on DFSV which makes state changes will not be reflected in the superclasses of the original objects — because the associated original objects are totally unrelated to one another.

The position *should* be as shown on the right — where the inheritance structures in the two worlds are totally unconnected — except at the initial explicit level within the instantiated proxy instance.

A related problem concerns inheritance from proxy classes. If class B now has a subclass, C, which is not wrapped, the correct behaviour of this subclass's methods is debatable. It is a simple enough matter to wrap the superclass and to invoke methods that C invokes on the (proxy) superclass on the matching original class. The difficulty occurs when one considers methods that C defines. Should C also be provided with a proxy? The SOM approach, which makes use of meta-objects, requires C to have a related metaclass which is a subclass of the metaclass related to B [7]. They thus require any subclasses of 'wrapped' classes to also be 'wrapped'. These types of issues are not trivial to solve and JavaCloak has yet to come up with a satisfactory solution.

4.2 Self and Encapsulation

The JavaCloak approach implements the proxy class and the wrapped class separately. This leads to two problems, the *self* [13] and the encapsulation problem.

The self problem arises because the meaning of self (or `this` in Java programs) is different in the proxy and in the wrapped instances. Thus the original object could instantiate a new instance of the same class. This object would not have a matching proxy object and therefore the behavioural modification being applied by the proxy would not be applied to this newly instantiated object. Lieberman [13] argues that inheritance-based languages such a Java cannot be used to implement delegation, which is, in essence, what a proxy does. If the programmer needs to intercept accesses to *all* instances of the original class this limitation is a problem, but if one wishes merely to intercept all accesses by the *client* program the proxy approach does not present a problem [17].

If the original object returns a reference to itself (`this`), or to another instance of the same class, to the proxy, this will be intercepted by the runtime manager. If a matching proxy already exists the proxy instance will be substituted, and if not, a proxy instance of the return value will be instantiated and returned to the proxy. This mechanism works when simple references to wrapped objects are passed back to the caller, but when a direct reference to an original object is embedded in another object that JavaCloak has no control over, then the situation is not solvable: the reference may be private which means JavaCloak cannot access it to perform the required conversion.

Hence in JavaCloak it is possible for a direct reference to an original instance to be passed across the boundary, thus breaking the proxy model. This occurs because a logical proxy model is being used and Java does not allow the redefinition of the meaning of `this` in the original object.

4.3 Identity and Equality

Any existing object-identity operation will work in JavaCloak because the identity of the two proxies will be compared, rather than the two original objects. However, if the reference to the proxy is passed to a service that follows the graph of objects reachable from the proxy, as Java's object serialization does, then the proxy and the real object will be serialized, which the programmer may not intend.

Constructor		draw		moveX		getSquares	
0.0058	0.2799	0.0264	0.0872	0.0278	0.1116	0.0299	0.3905

Table 1: Overhead of Calling Through a JavaCloak Proxy

In JavaCloak, all invocations on the public methods of an object are intercepted at the proxy, including the `hashCode` and `equals` methods defined on `java.lang.Object`. Thus, in JavaCloak, it is not possible to perform these operations on the proxies themselves as the `hashCode` and `equals` methods are forwarded to the original instance. This means that, at the JavaCloak implementation level, we cannot make use of these methods to manage the proxy. However, the JavaCloak management makes use of a `java.util.Hashtable` which needs to be able to call the `hashCode` and `equals` methods on the proxy. This requires additional objects to be registered with the lookup mechanism which then operate as a *placeholder* for the proxy when performing a lookup on it, via the hash table. The objects themselves cannot be used, so these tokens represent the real objects in this case.

When calling forward on the `equals` method it is necessary to translate the call from an operation on two proxy objects (`this` and the argument to `equals`), into an operation on two original objects. This is made possible by performing a proxy to original lookup in the JavaCloak runtime manager.

4.4 Transparency

Welch and Stroud [17] cite a number of difficulties inherent in the transparent addition of non-functional requirements by means of reflection — one of which is the handling of exceptions. There are two aspects to be considered:

1. Certain exceptions are declared in the method headers and are therefore 'expected' by the application. One may wish to intercept these exceptions for the purposes of better reporting or more standardised exception handling. Welch and Stroud [17] note that some researchers feel that this type of action allows adaptive runtime redefinition of exceptional behaviour [3]. Welch and Stroud argue that exception handling should not be re-definable but acknowledge the need for it in certain situations such as distributed exception handling.

2. A bigger problem for JavaCloak is that 'unexpected' exceptions may be thrown — caused by the reflection mechanism. These exceptions need to be handled in some consistent way, both in order to minimise disruption of the application, and so that the reflection system developers are apprised of the problem in case the exception was caused by a bug in the reflection code.

5 Performance Results

This section describes the typical runtime performance overhead of using a JavaCloak proxy. The timing results were taken on a lightly loaded Pentium II with version 1.3 of Java 2 from Sun Microsystems. The original class that was wrapped is given in Figure 4[10] and a proxy for it was generated by JavaCloak. A small test program was written that created a thousand instances of `Square` and invoked methods `draw`, `moveX` and `getSquares` on each. This program was run twenty times, both with and without the proxies, and an average taken. All the figures given are in seconds for one method invocation or one call to the constructor. The times for the proxy are on the right of the pair in Table 1.

```
public class Square {
  public  Square() {}

  public void draw() {}

  public void moveX(int delta, Square[][] sq)
     throws NullPointerException {}

  public Square [][] getSquares() {}
```

Figure 4: Original Class Used to Collect Performance Results

It is obvious from the table that JavaCloak proxies introduce a substantial overhead. In the constructor this is due to reflecting over the methods, and registering the proxy and the actual object with the JavaCloak runtime mechanism.

The overhead imposed on the invocations stems from needing to track the relationship between the proxy object and the wrapped object. For example, when an instance of `Square` is passed back from itself by the `getSquares` method, we need to map this value to its corresponding proxy. Currently, this is performed by means of a runtime lookup by the JavaCloak runtime manager.

This section should not be taken as evidence that the idea of proxies in general is bad or inefficient. The

[10] The implementation of the methods has been elided to save space as this is not relevant to this measurement.

performance overhead presented above is for one particular implementation of JavaCloak and there is room for improvement in these figures.

6 Conclusions and Future Work

Welch and Stroud [19] report that their Kava bytecode rewriting mechanism currently doubles the cost of instructions. The work reported in this paper has attempted to determine whether the proxy mechanism is a viable competitor to bytecode rewriting. It is clear in the current implementation of JavaCloak that the cost associated with statically-generated JavaCloak proxies is even *higher*, however there is room for improvement in the implementation.

The limiting factor is not so much the performance overhead, but the fact that Java does not allow the implementors of JavaCloak to seamlessly integrate their code with the code of someone using JavaCloak. This is primarily due to not being achieve the necessary flexibility when working at the source-code level.

Acknowledgement

I acknowledge the contributions of Huw Evans at the University of Glasgow. His collaboration and insights are greatly appreciated.

References

[1] Java 2 Platform, Standard Edition online documentation. `http://java.sun.com/j2se/1.3/docs/api/index.html/`, April 2000. Web Document.

[2] S. Chiba. Load-time structural reflection in java. In *14th European Conference on Object-Oriented Programming. ECOOP 2000*, pages 313–336, Sophia Antipolis and Cannes, France., 12 – 16 June 2000.

[3] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.

[5] E. Corporation. Objectstore Enterprise Edition Homepage. `http://www.object-store.net/objectstore/`, May 2000. Web Document.

[6] M. Dahm. Byte Code Engineering. In *Proceedings of JIT'99.*, Duesseldorf, Germany, 1999.

[7] S. Danforth and I. R. Forman. Reflections on Metaclass Programming in SOM. *ACM SIGPLAN Notices*, 29(10):440–440, Oct. 1994.

[8] J. de O Guimarães. Reflection for Statically Typed Languages. In *ECOOP'98*, Brussels, Belgium, July 20–24 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[10] M. Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, Erlangen, 1997.

[11] R. Keller and U. Hölzle. Binary Component Adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[12] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[13] H. Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In N. Meyrowitz, editor, *OOPSLA'86 Conference Proceedings: Object Oriented Programmng Systems, Languages and Applications.*, pages 214–223. ACM, 1986.

[14] H. Ogawa, K. S. S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. *Lecture Notes in Computer Science*, 1850:362–382, 2000.

[15] A. Oliva and L. E. Bizato. The Design and Implementation of Guarana. In *COOTS'99 — Proceedings of USENIX Conference on Object-Oriented Technology*, San Deigo, California, USA, May 3–7 1999.

[16] K. Renaud and H. Evans. Javacloak: Engineering Java Proxy Objects using Reflection. In M. Weber, editor, *NET.OBJECTDAYS 2000. Messekongresszentrum Erfurt, Germany*, October 9-12 2000.

[17] I. Welch and R. Stroud. Dalang — A Reflective Extension for Java. Technical Report CS-TR-672, Computing Science Department, University of Newcastle Upon Tyne, September 1999.

[18] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. *Lecture Notes in Computer Science*, 1616:2–21, 1999.

[19] I. Welch and R. J. Stroud. Kava — Using Byte code Rewriting to add Behavioural Reflection to Java. In *COOTS '01 — Proceedings of USENIX Conference on Object-Oriented Technology*, San Antonio, Texas, USA, January 29 – February 2 2001.

[20] Z. Wu. Reflective Java and a Reflective-Component-Based Transaction Architecture. In J. C. Fabre and S. Chiba, editors, *Proceedings of OOOPSLA'98. Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 18–22 October 1998.